16丨案例：性能监控工具之Grafana+Prometheus+Exporters

2020-01-20 高楼

性能测试实战30讲 进入课程 >

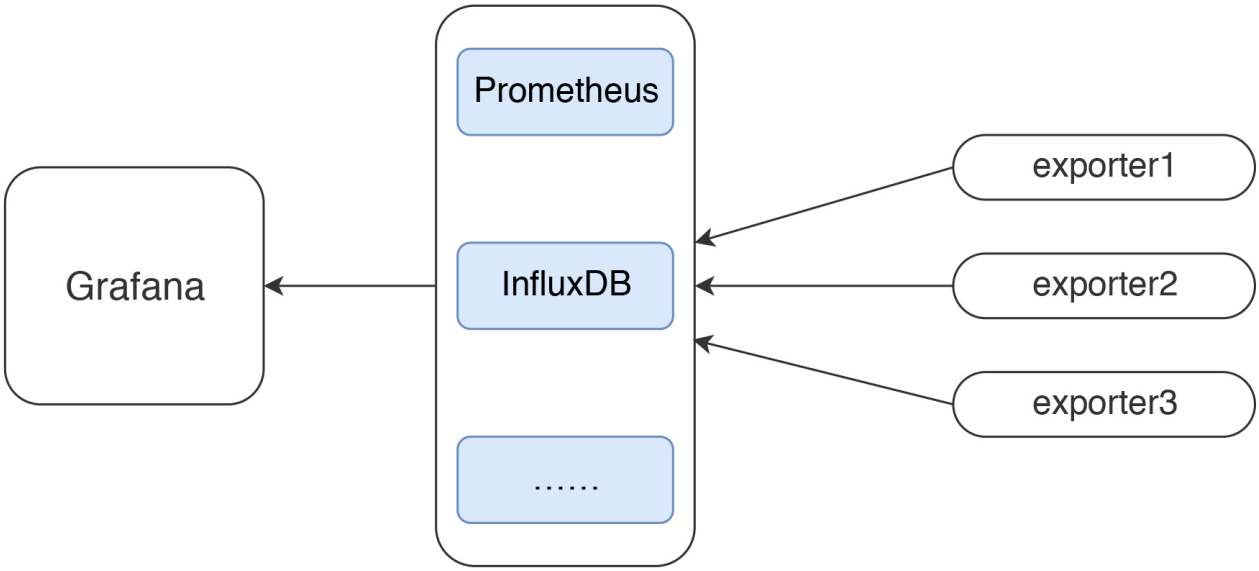
讲述：高楼

时长 14:27　大小 13.25M

在本模块中，我将把几个常用的监控部分给梳理一下。前面我们提到过，在性能监控图谱中，有操作系统、应用服务器、中间件、队列、缓存、数据库、网络、前端、负载均衡、Web 服务器、存储、代码等很多需要监控的点。

显然这些监控点不能在一个专栏中全部覆盖并一一细化，我只能找最常用的几个，做些逻辑思路的说明，同时也把具体的实现描述出来。如果你遇到了其他的组件，也需要一一实现这些监控。

在本篇中，主要想说明白下图的这个监控逻辑。



这应该是现在最流行的一套监控逻辑了吧。

我今天把常见的使用 Grafana、Prometheus、InfluxDB、Exporters 的数据展示方式说一下，如果你刚进入性能测试领域，也能有一个感性的认识。

有测试工具，有监控工具，才能做后续的性能分析和瓶颈定位，所以有必要把这些工具的逻辑跟你摆一摆。

所有做性能的人都应该知道一点，不管数据以什么样的形式展示，最要紧的还是看数据的来源和含义，以便做出正确的判断。

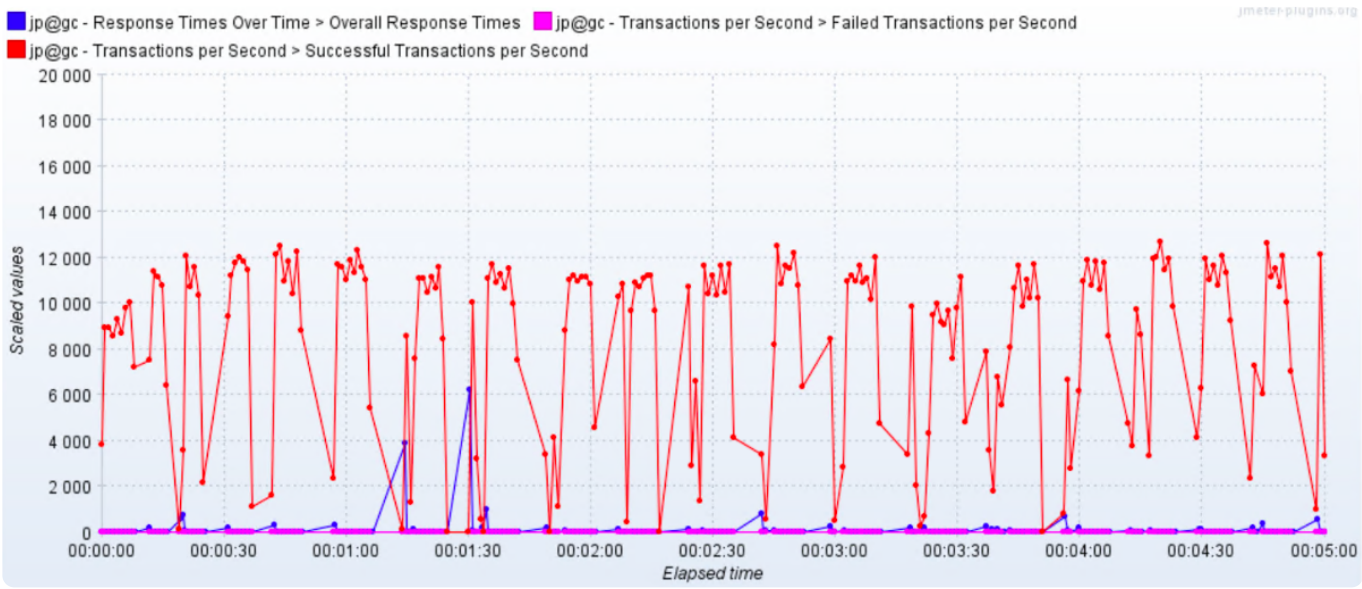
我先说明一下 JMeter 和 node_exporter 到 Grafana 的数据展示逻辑。至于其他的 Exporter，我就不再解释这个逻辑了，只说监控分析的部分。

JMeter+InfluxDB+Grafana 的数据展示逻辑

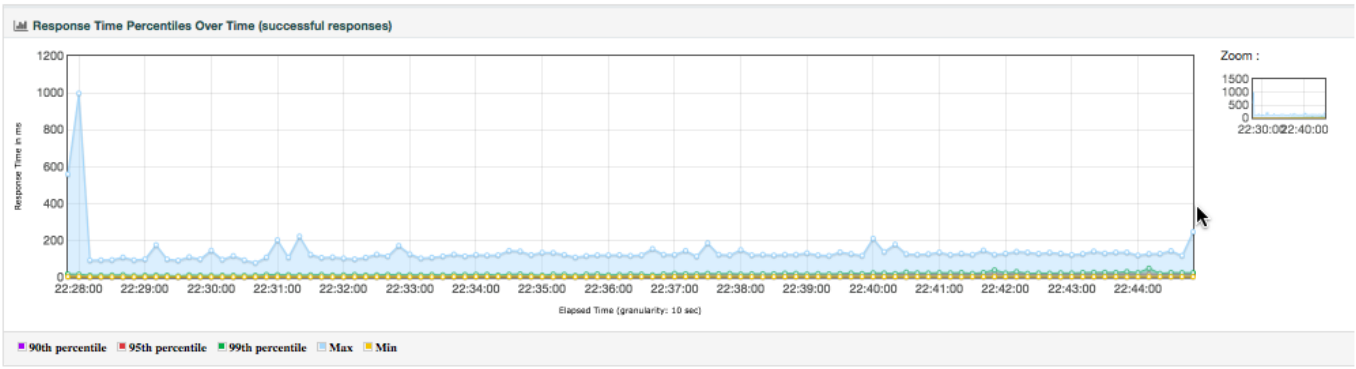
一般情况下，我们用 JMeter 做压力测试时，都是使用 JMeter 的控制台来查看结果。如下图所示：

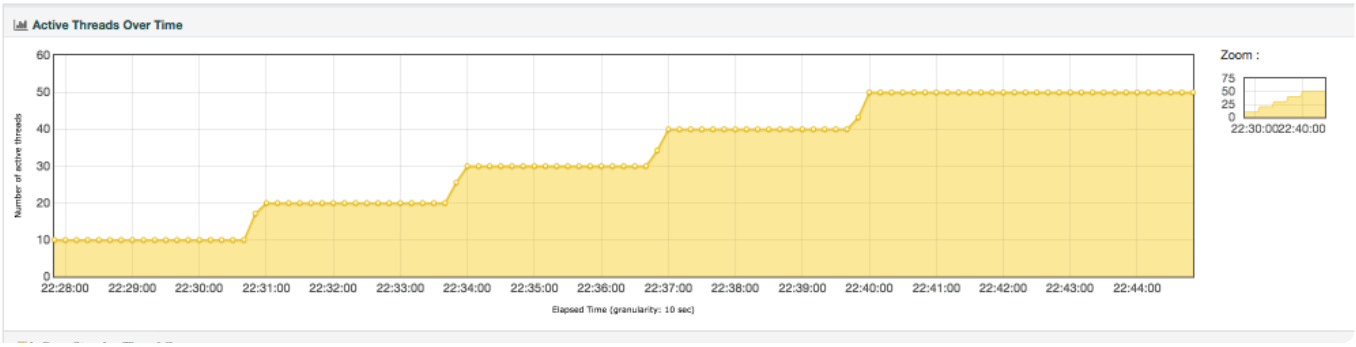
```
summary +    656 in 00:00:12 =   56.2/s Avg:   115 Min:    22 Max:  1043 Err:     0 (0.00%) Active: 641 Started: 641 Finished: 0
summary +    493 in 00:00:30 =   16.3/s Avg:   802 Min:    19 Max: 31294 Err:    27 (5.48%) Active: 973 Started: 1000 Finished: 27
summary =   1149 in 00:00:42 =   27.5/s Avg:   410 Min:    19 Max: 31294 Err:    27 (2.35%)
summary +    105 in 00:00:30 =    3.5/s Avg:    68 Min:    18 Max:  1169 Err:     0 (0.00%) Active: 973 Started: 1000 Finished: 27
summary =   1254 in 00:01:12 =   17.5/s Avg:   382 Min:    18 Max: 31294 Err:    27 (2.15%)
summary +    102 in 00:00:31 =    3.3/s Avg:    52 Min:    18 Max:   480 Err:     0 (0.00%) Active: 973 Started: 1000 Finished: 27
summary =   1356 in 00:01:43 =   13.2/s Avg:   357 Min:    18 Max: 31294 Err:    27 (1.99%)
summary +    100 in 00:00:29 =    3.4/s Avg:    78 Min:    19 Max:   751 Err:     0 (0.00%) Active: 973 Started: 1000 Finished: 27
summary =   1456 in 00:02:12 =   11.0/s Avg:   338 Min:    18 Max: 31294 Err:    27 (1.85%)
summary +     97 in 00:00:30 =    3.2/s Avg:    53 Min:    19 Max:   469 Err:     0 (0.00%) Active: 973 Started: 1000 Finished: 27
summary =   1553 in 00:02:43 =    9.6/s Avg:   320 Min:    18 Max: 31294 Err:    27 (1.74%)
summary +    100 in 00:00:29 =    3.4/s Avg:   110 Min:    20 Max:   938 Err:     0 (0.00%) Active: 973 Started: 1000 Finished: 27
summary =   1653 in 00:03:12 =    8.6/s Avg:   307 Min:    18 Max: 31294 Err:    27 (1.63%)
summary +     92 in 00:00:33 =    2.8/s Avg:    70 Min:    20 Max:  1305 Err:     0 (0.00%) Active: 973 Started: 1000 Finished: 27
summary =   1745 in 00:03:44 =    7.8/s Avg:   295 Min:    18 Max: 31294 Err:    27 (1.55%)
summary +    100 in 00:00:27 =    3.6/s Avg:    75 Min:    18 Max:   544 Err:     0 (0.00%) Active: 973 Started: 1000 Finished: 27
summary =   1845 in 00:04:12 =    7.3/s Avg:   283 Min:    18 Max: 31294 Err:    27 (1.46%)
summary +    101 in 00:00:31 =    3.3/s Avg:    81 Min:    18 Max:   500 Err:     0 (0.00%) Active: 973 Started: 1000 Finished: 27
summary =   1946 in 00:04:43 =    6.9/s Avg:   272 Min:    18 Max: 31294 Err:    27 (1.39%)
summary +    695 in 00:00:29 =   24.0/s Avg: 160616 Min:    18 Max: 300303 Err:     0 (0.00%) Active: 368 Started: 1000 Finished: 632
summary =   2641 in 00:05:12 =    8.5/s Avg: 42468 Min:    18 Max: 300303 Err:    27 (1.02%)
```

或者装个插件来看结果：



或者用 JMeter 来生成 HTML：





这样看都没有问题，我们在前面也强调过，对于压力工具来说，我们最多只关心三条曲线的数据：TPS（T 由测试目标定义）、响应时间、错误率。这里的错误率还只是辅助排查问题的曲线，没有问题时，只看 TPS 和响应时间即可。

不过采取以上三种方式有几个方面的问题。

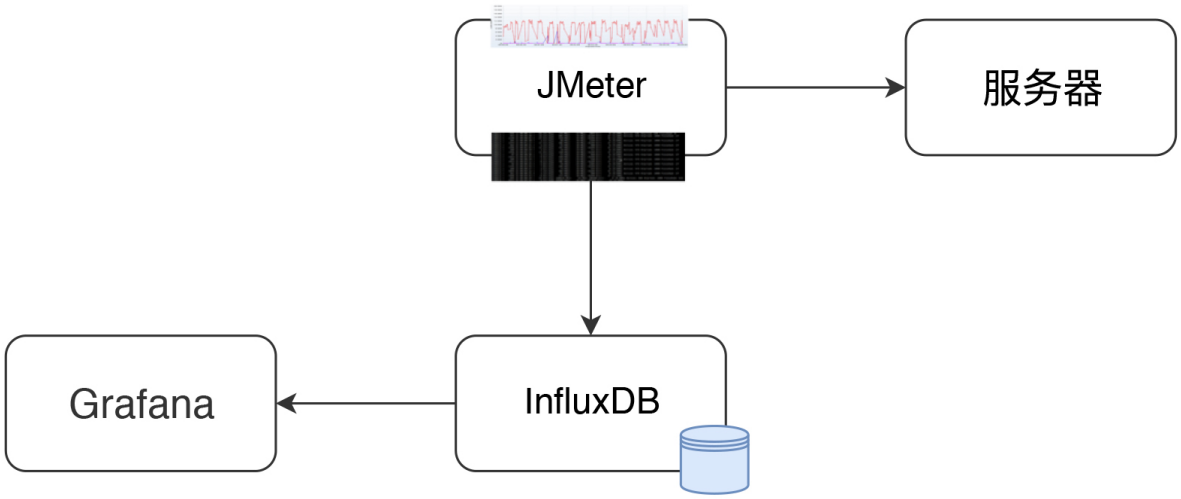
1. 整理结果时比较浪费时间。
2. 在 GUI 用插件看曲线，做高并发时并不现实。
3. 在场景运行时间比较长的时候，采用生成 HTML 的方式，会出现消耗内存过大的情况，而实际上，在生成的结果图中，有很多生成的图我们并不是那么关注。
4. 生成的结果保存之后再查看比较麻烦，还要一个个去找。

那么如何解决这几个问题呢？

用 JMeter 的 Backend Listener 帮我们实时发送数据到 InfluxDB 或 Graphite 可以解决这样的问题。Graphite Backend Listener 的支持是在 JMeter 2.13 版本，InfluxdDB Backend Listener 的支持是在 JMeter 3.3 的版本，它们都是用异步的方式把数据发送出来，以便查看。

其实有这个 JMeter 发送给 InfluxDB 的数据之后，我们不需要看上面的那些 HTML 数据，也可以直观地看到系统性能的性能趋势。并且这样保存下来的数据，在测试结束后想再次查看也比较方便比对。

JMeter+InfluxDB+Grafana 的结构如下：



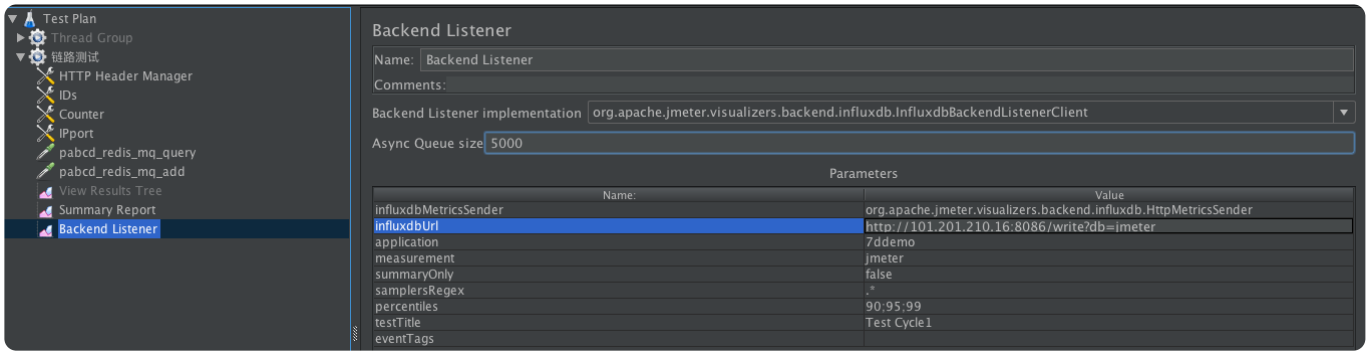
在这个结构中，JMeter 发送压力到服务器的同时，统计下 TPS、响应时间、线程数、错误率等信息。默认每 30 秒在控制台输出一次结果（在 jmeter.properties 中有一个参数 #summariser.interval=30 可以控制）。配置了 Backend Listener 之后，将统计出的结果异步发送到 InfluxDB 中。最后在 Grafana 中配置 InfluxDB 数据源和 JMeter 显示模板。

然后就可以实时查看 JMeter 的测试结果了，这里看到的数据和控制台的数据是一样。

但如果这么简单就说完了，这篇文章也就没价值了。下面我们来说一下，数据的传输和展示逻辑。

JMeter 中 Backend Listener 的配置

下面我们就 InfluxDB 的 Backend Listener 做个说明。它的配置比较简单，在脚本中加上即可。



我们先配置好 influxdb Url、application 等信息，application 这个配置可以看成是场景名。

那么 JMeter 如何将数据发给 InfluxDB 呢？请看源码中的关键代码，如下所示：

复制代码

```
    private void addMetrics(String transaction, SamplerMetric metric) {
        // FOR ALL STATUS
        addMetric(transaction, metric.getTotal(), metric.getSentBytes(), metri
                metric.getAllMaxTime(), allPercentiles.values(), metric::getAl
        // FOR OK STATUS
        addMetric(transaction, metric.getSuccesses(), null, null, TAG_OK, metr
                metric.getOkMaxTime(), okPercentiles.values(), metric::getOkPe
        // FOR KO STATUS
        addMetric(transaction, metric.getFailures(), null, null, TAG_KO, metri
                metric.getKoMaxTime(), koPercentiles.values(), metric::getKoPe


        metric.getErrors().forEach((error, count) -> addErrorMetric(transactio
                    error.getResponseMessage(), count));
    }

```

从这段代码可以看出，站在全局统计的视角来看，这里把 JMeter 运行的统计结果，比如事务的 Total 请求、发送接收字节、平均值、最大值、最小值等，都加到 metric 中，同时也会把成功和失败的事务信息添加到 metric 中去。

在源码中，还有更多的添加 metric 的步骤，你有兴趣的话，也可以看一下 JMeter 源码中的`InfluxdbBackendListenerClient.java`。

保存了 metric 之后，再使用 InfluxdbMetricsSender 发送到 Influxdb 中去。发送关键代码如下：

复制代码

```
  @Override
   public void writeAndSendMetrics() {
........
       if (!copyMetrics.isEmpty()) {
           try {
               if(httpRequest == null) {
                   httpRequest = createRequest(url);
               }
               StringBuilder sb = new StringBuilder(copyMetrics.size()*35);
               for (MetricTuple metric : copyMetrics) {
                   // Add TimeStamp in nanosecond from epoch ( default in Inf
                   sb.append(metric.measurement)
                       .append(metric.tag)
                       .append(" ") //$NON-NLS-1$
                       .append(metric.field)
                       .append(" ")
                       .append(metric.timestamp+"000000")
                       .append("\n"); //$NON-NLS-1$
               }


               StringEntity entity = new StringEntity(sb.toString(), Standard

               httpRequest.setEntity(entity);
               lastRequest = httpClient.execute(httpRequest, new FutureCallba
                   @Override
                   public void completed(final HttpResponse response) {
                       int code = response.getStatusLine().getStatusCode();
                       /*
                        * HTTP response summary 2xx: If your write request re
                        * HTTP 204 No Content, it was a success! 4xx: InfluxD
                        * could not understand the request. 5xx: The system i
                        * overloaded or significantly impaired.
                        */
                       if (MetricUtils.isSuccessCode(code)) {
                           if(log.isDebugEnabled()) {
                               log.debug("Success, number of metrics written:
                           }
                       } else {
                           log.error("Error writing metrics to influxDB Url:
                       }
                   }
                   @Override
                   public void failed(final Exception ex) {
                       log.error("failed to send data to influxDB server : {}
                   }
                   @Override
                   public void cancelled() {
                       log.warn("Request to influxDB server was cancelled");
                   }
```

```
                });
 ........
            }
        }
    }
```

通过 writeAndSendMetrics，就将所有保存的 metrix 都发给了 InfluxDB。

InfluxDB 中的存储结构

然后我们再来看下 InfluxDB 中如何存储：

复制代码

```
> show databases
name: databases
name
----
_internal
jmeter
> use jmeter
Using database jmeter
>
> show MEASUREMENTS
name: measurements
name
----
events
jmeter
> select * from events where application='7ddemo'
name: events
time                application text                title
----                ----------- ----                -----
1575255462806000000 7ddemo      Test Cycle1 started ApacheJMeter
1575256463820000000 7ddemo      Test Cycle1 ended   ApacheJMeter
..............
n> select * from jmeter where application='7ddemo' limit 10
name: jmeter
time                application avg                count countError endedT hit
----                ----------- ---                ----- ---------- ------ ---
1575255462821000000 7ddemo                                         0
1575255467818000000 7ddemo      232.82352941176472 17    0                17
1575255467824000000 7ddemo      232.82352941176472 17
1575255467826000000 7ddemo      232.82352941176472 17
1575255467829000000 7ddemo                                         0
1575255472811000000 7ddemo      205.4418604651163  26    0                26
1575255472812000000 7ddemo                                         0
1575255472812000000 7ddemo      205.4418604651163  26
```

```
1575255472812000000 7ddemo      205.4418604651163  26
1575255477811000000 7ddemo      198.2142857142857  27    0                27
```

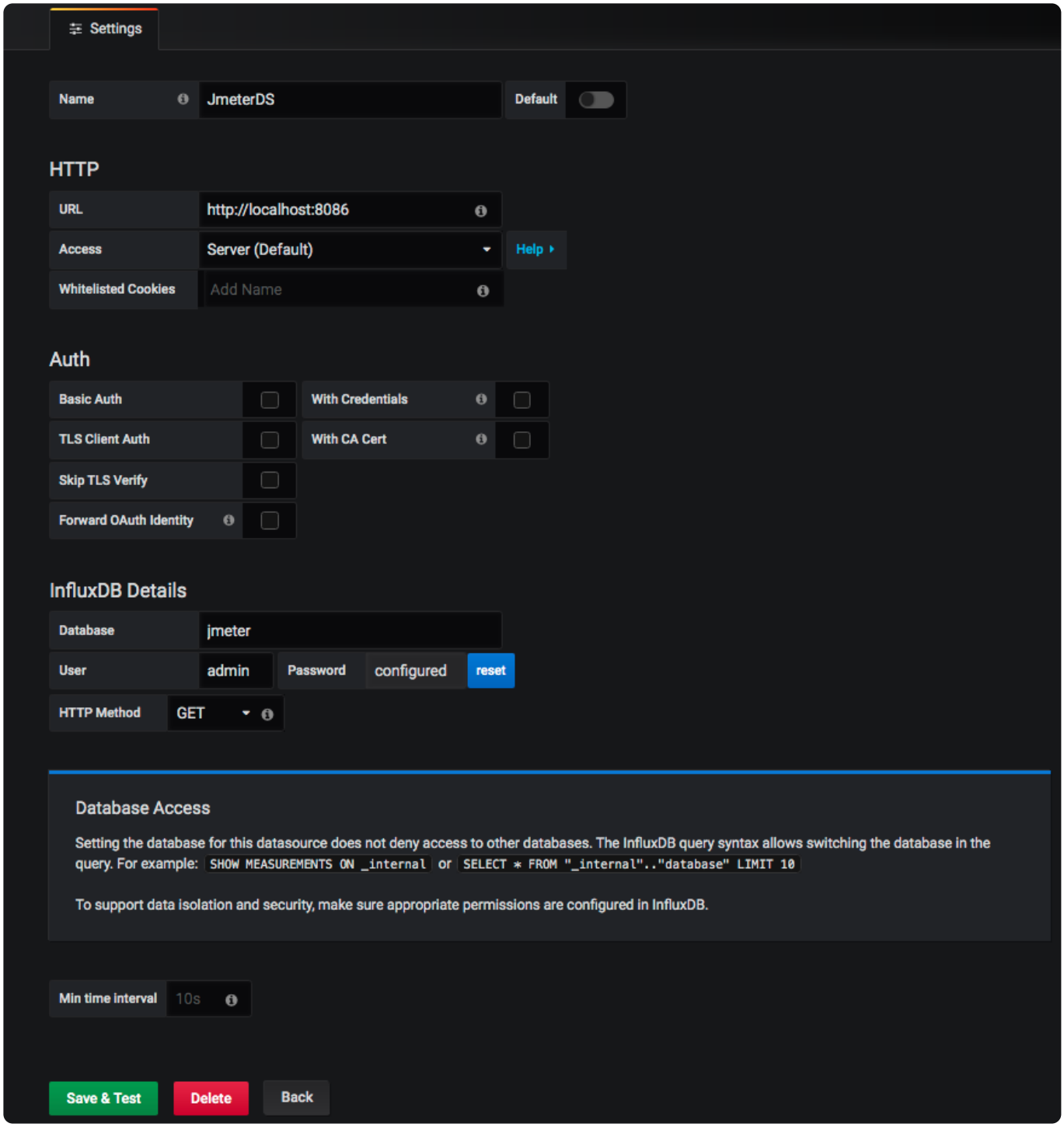
这段代码也就是说，在 InfluxDB 中，创建了两个 MEASUREMENTS，分别是 events 和 jmeter。这两个各自存了数据，我们在界面中配置的 testtile 和 eventTags 放在了 events 这个 MEASUREMENTS 中。在模板中这两个值暂时都是不用的。

在 jmeter 这个 MEASUREMENTS 中，我们可以看到 application 和事务的统计信息，这些值和控制台一致。

在 Grafana 中显示的时候，就是从这个表中取出的数据，根据时序做的曲线。

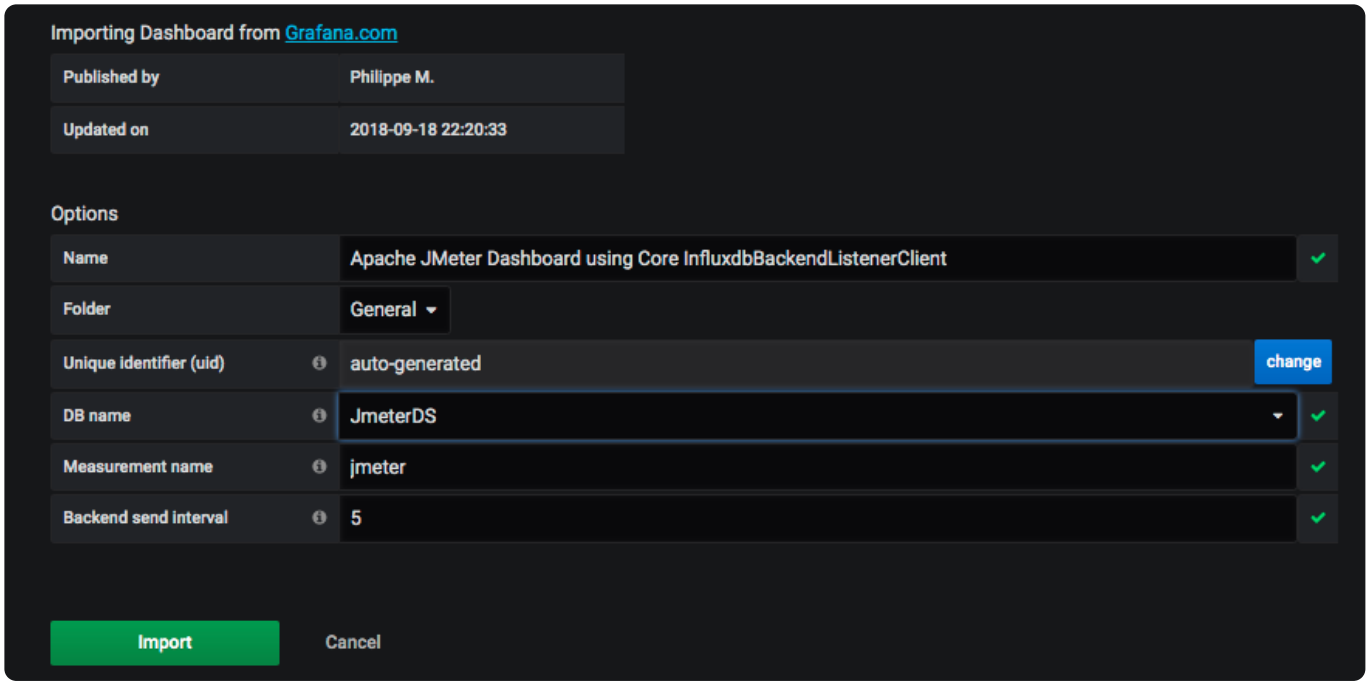
Grafana 中的配置

有了 JMeter 发送到 InfluxDB 中的数据，下面就来配置一下 Grafana 中的展示。首先，要配置一个 InfluxDB 数据源。如下所示：



在这里配置好 URL、Database、User、Password 之后，直接点击保存即可。

然后添加一个 JMeter dashboard，我们常用的 dashboard 是 Grafana 官方 ID 为 5496 的模板。导入进来后，选择好对应的数据源。



然后就看到界面了。

这时还没有数据，我们稍后做个示例，看下 JMeter 中的数据怎么和这个界面的数据对应起来。

我们先看下图中两个重要的数据查询语句吧。

TPS 曲线：

复制代码

```
SELECT last("count") / $send_interval FROM "$measurement_name" WHERE ("transac
```

上面这个就是 Total TPS 了，在这里称为 throughput。关于这个概念，我在第一篇中就已经有了说明，这里再次提醒，概念的使用在团队中要有统一的认识，不要受行业内一些传统信息的误导。

这里取的数据来自 MEASUREMENTS 中成功状态的所有事务。

响应时间曲线：

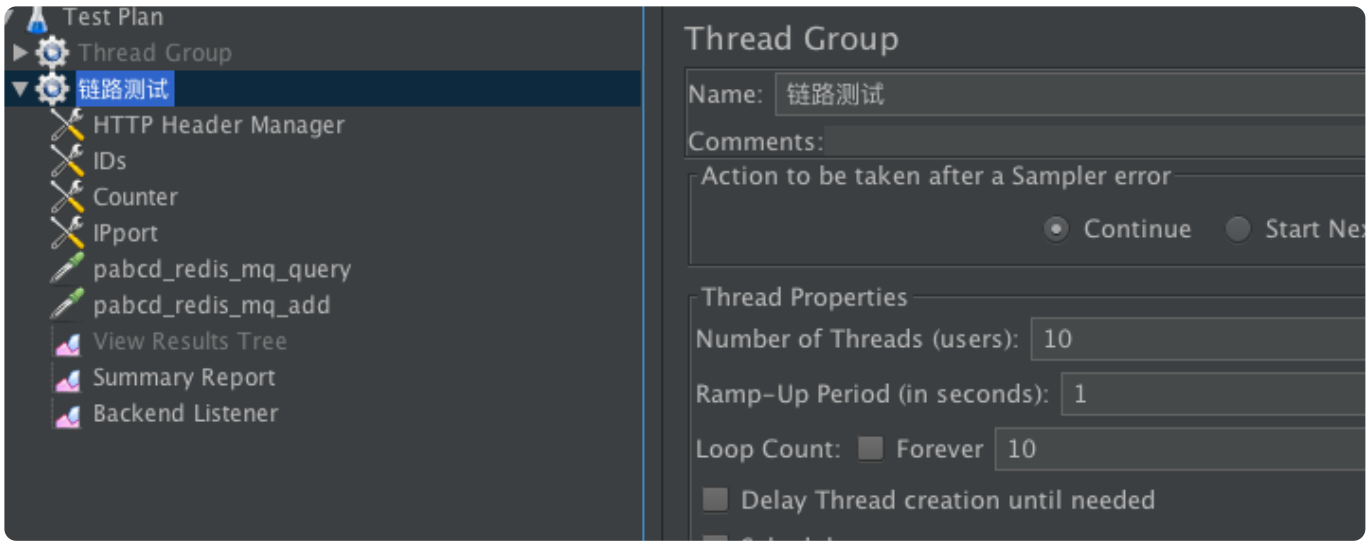
```
1  SELECT mean("pct95.0") FROM "$measurement_name" WHERE ("application" =~ /^$app
```

这里是用 95 pct 内的响应时间画出来的曲线。

整体展示出来的效果如下：

数据比对

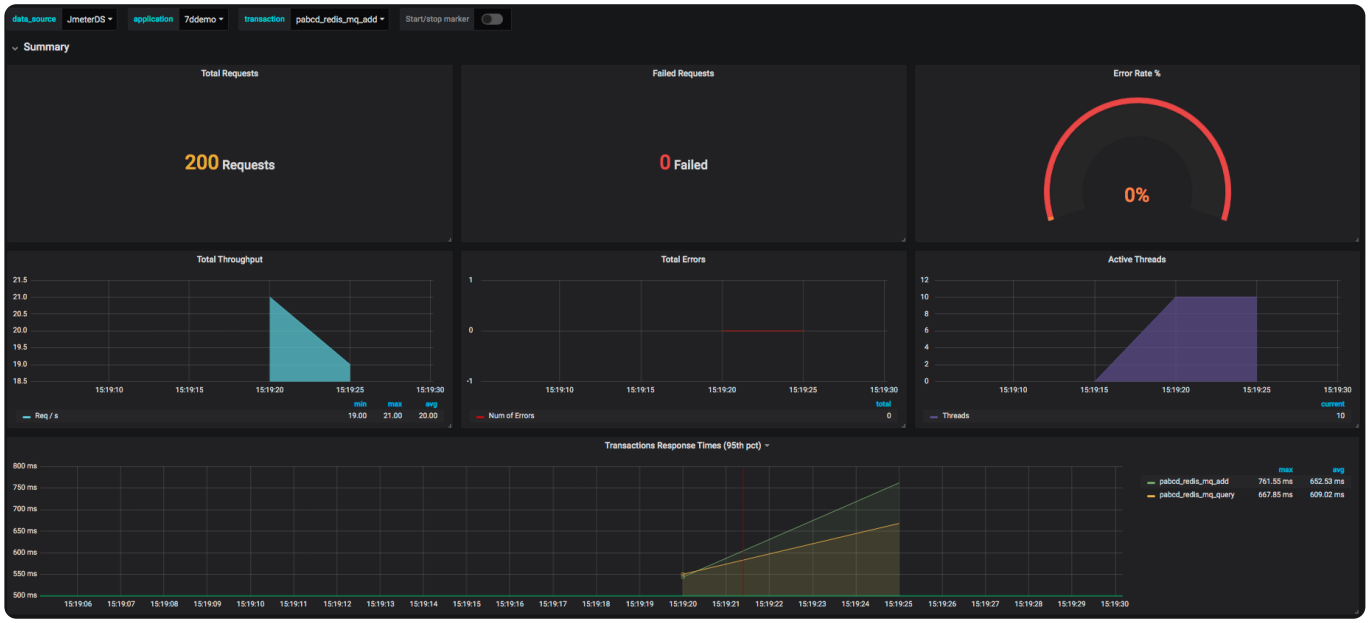
首先，我们在 JMeter 中配置一个简单的场景。10 个线程，每个线程迭代 10 次，以及两个 HTTP 请求。



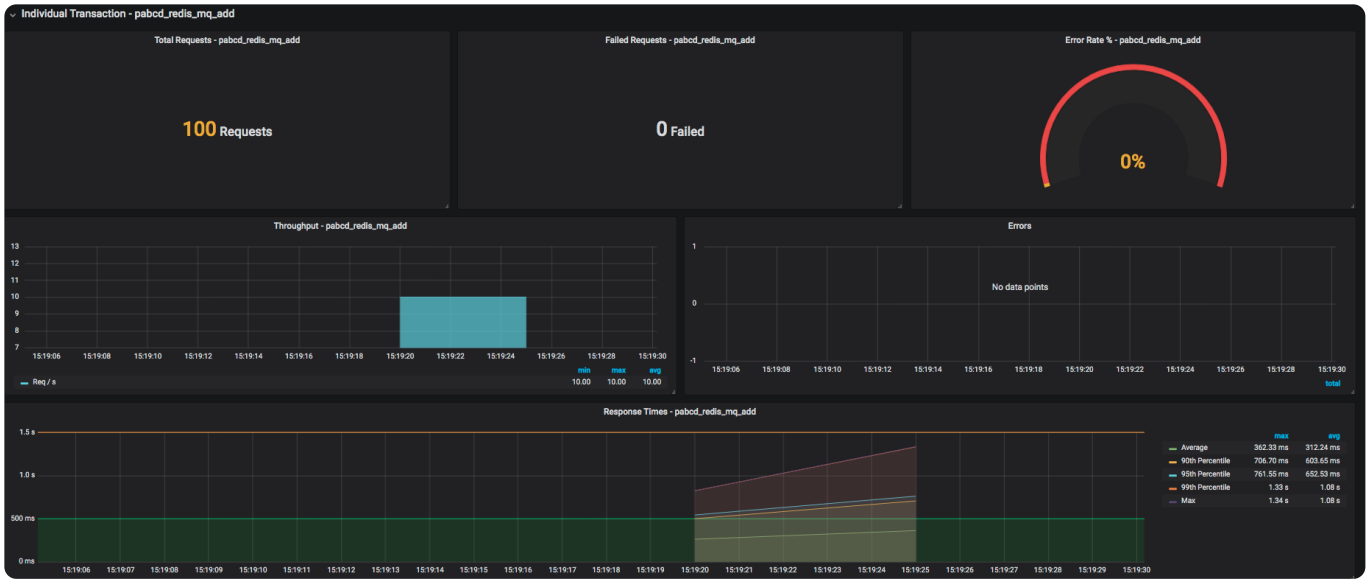
也就是说，这时会产生 10x10x2=200 次请求。我们用 JMeter 跑起来看一下。

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
pabcd_redis_mq_qu...	100	331	109	1171	184.23	0.00%	13.2/sec	5.15	2.70	400.8
pabcd_redis_mq_add	100	362	68	1337	231.37	0.00%	12.9/sec	2.25	5.05	178.0
TOTAL	200	346	68	1337	209.72	0.00%	25.4/sec	7.17	7.55	289.4

看到了吧，这个请求数和我们预想的一样。下面我们看一下 Grafana 中展示出来的结果。



还有针对每个事务的统计情况。

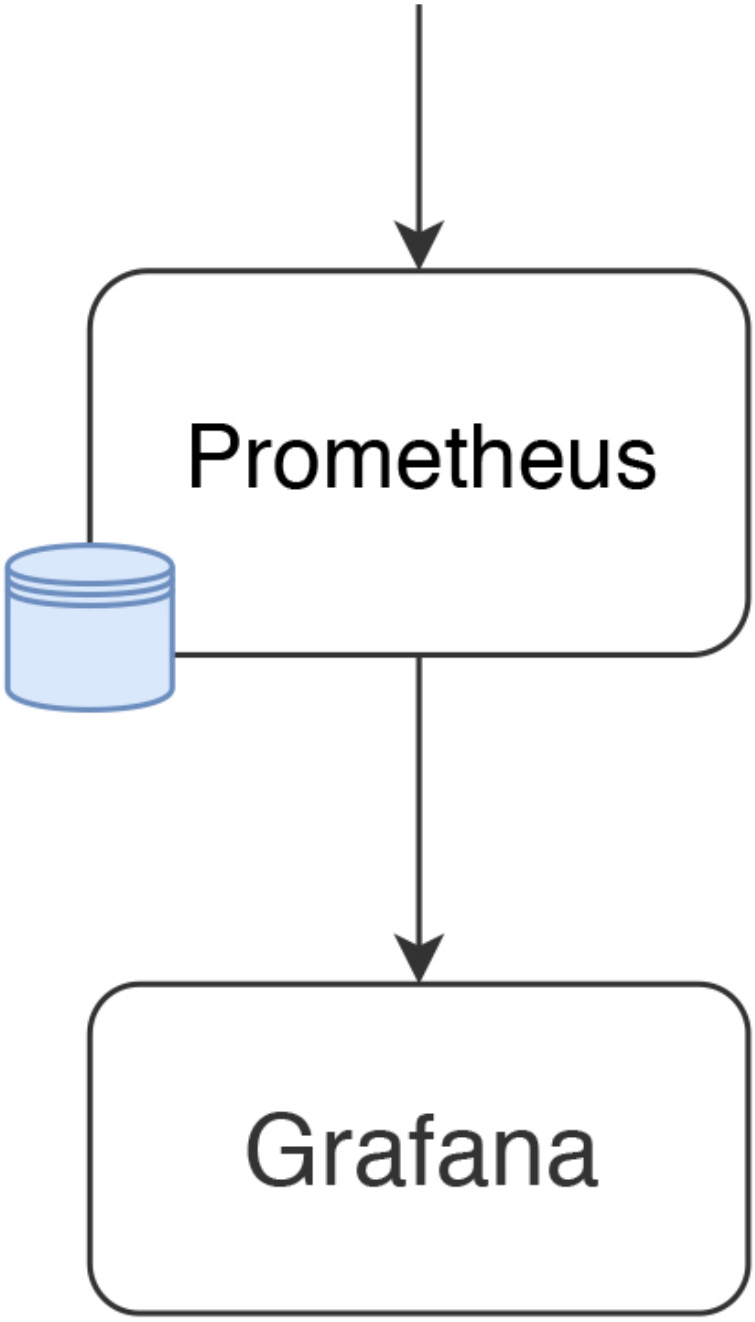


至此，JMeter 到 Grafana 的展示过程就完成了。以后我们就不用再保存 JMeter 的执行结果了，也不用等着 JMeter 输出 HTML 了。

node_exporter+Prometheus+Grafana 的数据展示逻辑

对性能测试来说，在常用的 Grafana+Prometheus+Exporter 的逻辑中，第一步要看的就是操作系统资源了。所以在这一篇中，我们将以 node_exporter 为例来说明一下操作系统抽取数据的逻辑，以便知道监控数据的来源，至于数据的含义，我们将在后续的文章中继续描述。

首先，我们还是要画一个图。



现在 node_exporter 可以支持很多个操作系统了。官方列表如下：

node_exporter

Exporter for machine metrics prometheus/node_exporter

0.18.1 / 2019-06-04 Release notes

File name	OS	Arch	Size	SHA256 Checksum
node_exporter-0.18.1.darwin-386.tar.gz	darwin	386	3.78 MiB	61a13b13f5a98bafd6e0dec17c6579acbc13f8a1e24a8e9206a8017edb248460
node_exporter-0.18.1.darwin-amd64.tar.gz	darwin	amd64	3.99 MiB	20fadb3108de0a9cc70a1333394e5be90416b4f91025f9fc66f5736335e94398
node_exporter-0.18.1.linux-386.tar.gz	linux	386	7.47 MiB	a6c7eb64bb5f27a5567d545a1b93780f3aa72d0627751fd9f054626bb542a4b5
node_exporter-0.18.1.linux-amd64.tar.gz	linux	amd64	7.71 MiB	b2503fd932f85f4e5baf161268854bf5d22001869b84f00fd2d1f57b51b72424
node_exporter-0.18.1.linux-arm64.tar.gz	linux	arm64	7.19 MiB	d5a28c46e74f45b9f2158f793a6064fd9fe8fd8da6e0d1e548835ceb7beb1982
node_exporter-0.18.1.linux-armv5.tar.gz	linux	armv5	7.18 MiB	1eecbaa2a7e58dc2a5b18e960c48842e5e158c9e2eea4d8a4ba32b98ca2f638a
node_exporter-0.18.1.linux-armv6.tar.gz	linux	armv6	7.18 MiB	6f3cb593c15c12cdfaef20d7e1c61d28ef822af6fc8c85d670cb3f0a1212778a
node_exporter-0.18.1.linux-armv7.tar.gz	linux	armv7	7.17 MiB	5de85067f44b42b22d62b2789cb1a379ff5559649b99795cd1ba0c144b512ad0
node_exporter-0.18.1.linux-mips.tar.gz	linux	mips	7.12 MiB	9ef7c932970bc823a63347c3cdd8a34a4ef9d327cd5513600435dfd74d046755
node_exporter-0.18.1.linux-mips64.tar.gz	linux	mips64	7.29 MiB	c2721c1b85e3024e61f37fb2dc44a57f6d4eed8cc0576185a1dedea20e36fb31
node_exporter-0.18.1.linux-mips64le.tar.gz	linux	mips64le	7.06 MiB	ae262af96dd7409aeefe28f8ea6cb1b003774448370 57ed67694d8fa1b75b848
node_exporter-0.18.1.linux-mipsle.tar.gz	linux	mipsle	6.97 MiB	40860be242f563e3e10972685f1d1654c9b5ca9686b26bde4a422f57a1ebdd18
node_exporter-0.18.1.linux-ppc64.tar.gz	linux	ppc64	7.37 MiB	b41f860dbe23b72cf2ae939dd6bb43ea3ddde268f5a964cf6f8d490fed1ed034
node_exporter-0.18.1.linux-ppc64le.tar.gz	linux	ppc64le	7.17 MiB	27996a62327e07041b5dd2f09d6054c7c21244e39358da5d9b44b96daf6a2bc0
node_exporter-0.18.1.linux-s390x.tar.gz	linux	s390x	7.82 MiB	0bc212b9db6c2201b2b38d46de2d4cc75b7f4648d7616a87d7616e85f0d6cba4

当然不是说只支持这些，你也可以扩展自己的 Exporter。

配置 node_exporter

node_exporter 目录如下：

复制代码

```
[root@7dgroup2 node_exporter-0.18.1.linux-amd64]# ll
total 16524
-rw-r--r-- 1 3434 3434    11357 Jun  5 00:50 LICENSE
-rwxr-xr-x 1 3434 3434 16878582 Jun  5 00:41 node_exporter
-rw-r--r-- 1 3434 3434      463 Jun  5 00:50 NOTICE
```

启动：

复制代码

```
[root@7dgroup2 node_exporter-0.18.1.linux-amd64]#./node_exporter --web.listen-
```

是不是很简洁？如果想看更多的功能，可以查看下它的帮助。

配置 Prometheus

先下载 Prometheus：

```
[root@7dgroup2 data]# wget -c https://github.com/prometheus/prometheus/release
.........
100%[=======================================================================

2019-11-29 15:40:16 (157 KB/s) - ‘prometheus-2.14.0.linux-amd64.tar.gz’ saved


[root@7dgroup2 data]

```

解压之后，我们可以看到目录结构如下：

```
[root@7dgroup2 prometheus-2.11.1.linux-amd64]# ll
total 120288
drwxr-xr-x. 2 3434 3434     4096 Jul 10 23:26 console_libraries
drwxr-xr-x. 2 3434 3434     4096 Jul 10 23:26 consoles
drwxr-xr-x. 3 root root     4096 Nov 30 12:55 data
-rw-r--r--. 1 3434 3434    11357 Jul 10 23:26 LICENSE
-rw-r--r--. 1 root root       35 Aug  7 23:19 node.yml
-rw-r--r--. 1 3434 3434     2770 Jul 10 23:26 NOTICE
-rwxr-xr-x. 1 3434 3434 76328852 Jul 10 21:53 prometheus
-rw-r--r--  1 3434 3434     1864 Sep 21 09:36 prometheus.yml
-rwxr-xr-x. 1 3434 3434 46672881 Jul 10 21:54 promtool
[root@7dgroup2 prometheus-2.11.1.linux-amd64]#
```

在`prometheus.yml`中添加如下配置，以取数据：

```
  - job_name: 's1'
    static_configs:
    - targets: ['172.17.211.143:9200']
```

启动：

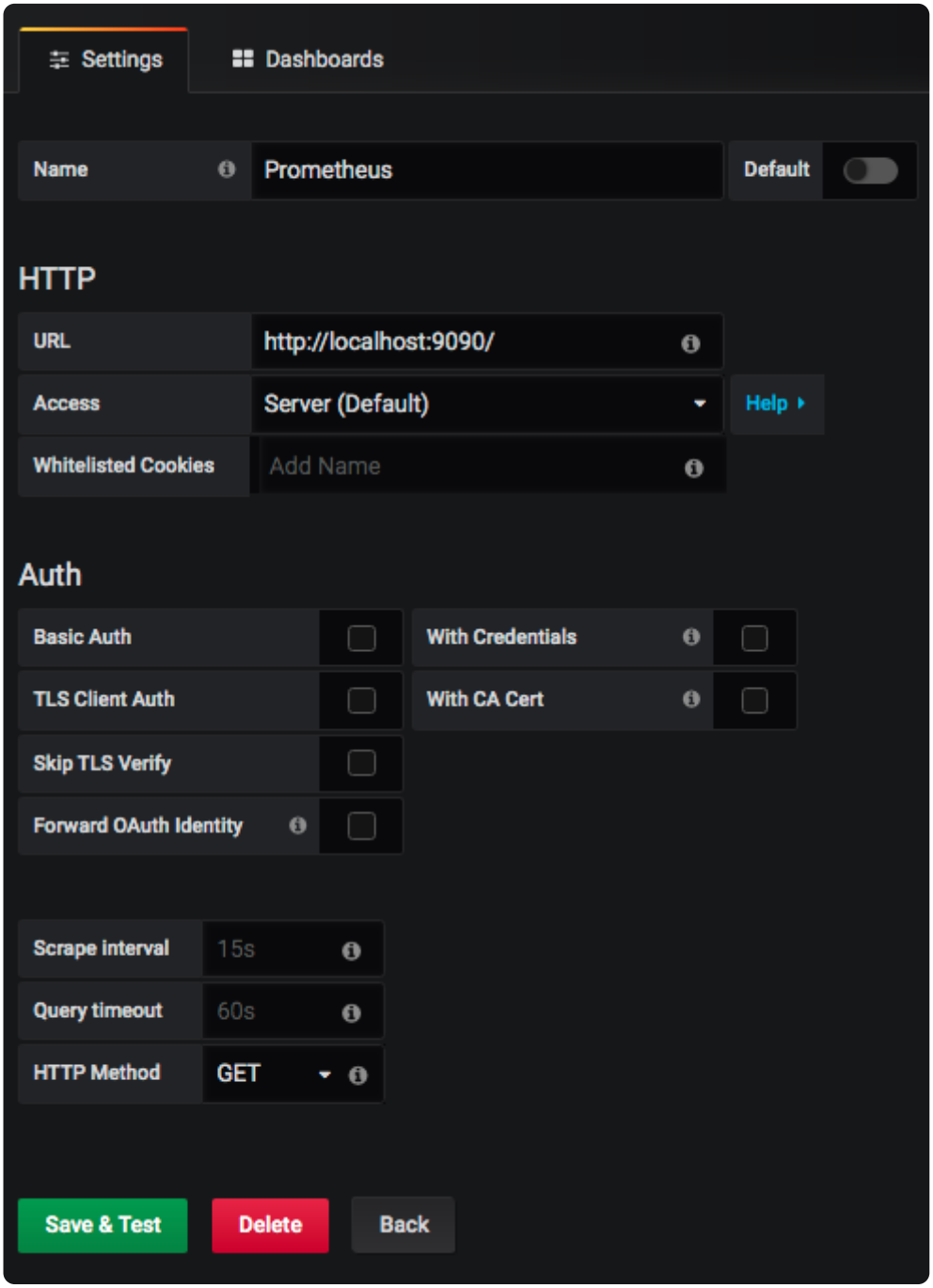
```
[root@7dgroup2 data]# ./prometheus --config.file=prometheus.yml &

```

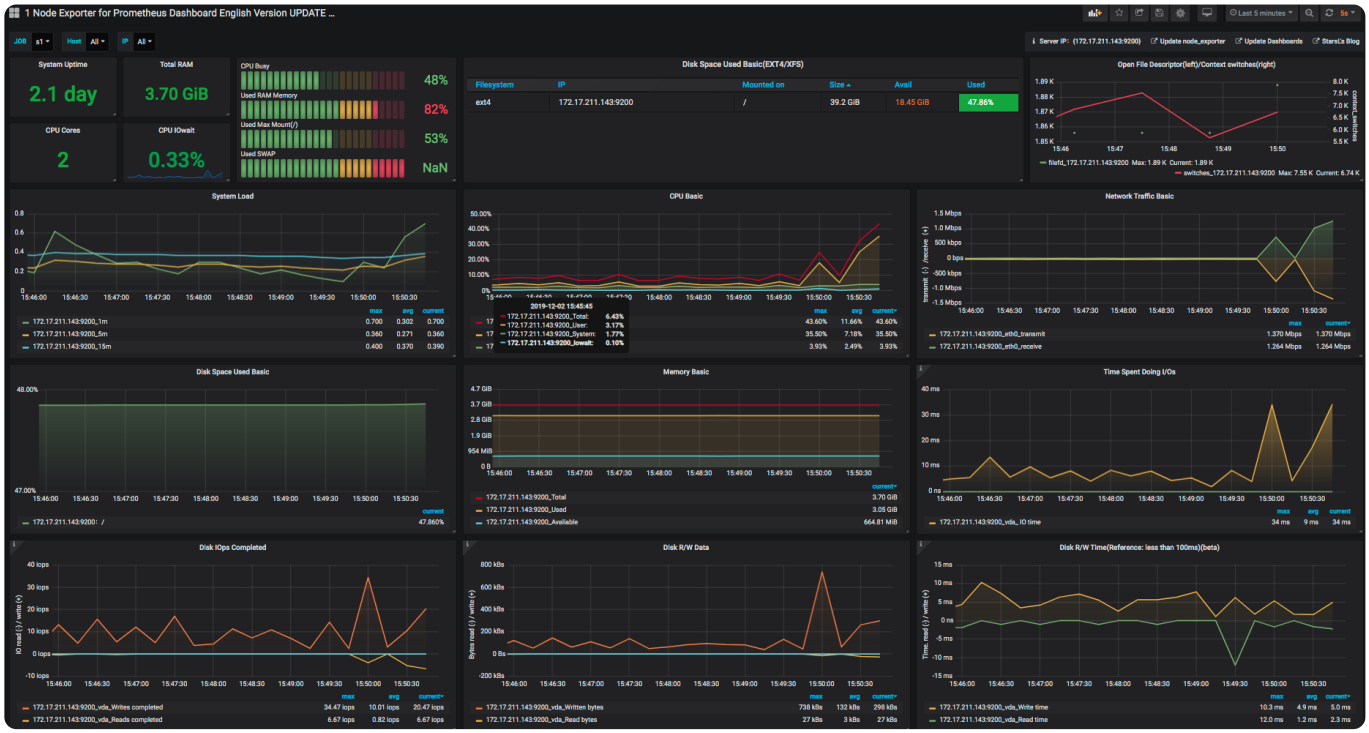
这样就行了吗？当然不是。根据上面的流程图，我们还需要配置 Grafana。

配置 Grafana

首先配置一个数据源，非常简单。如下所示：



再配置一个 node_exporter 的模板，比如我这里选择了官方模板（ID：11074），展示如下：



数据逻辑说明

说明完上面的过程之后，对我们做性能测试和分析的人来说，最重要的，就是要知道数据的来源和含义了。

拿上面图中的 CPU 使用率来说吧（因为 CPU 使用率是非常重要的一个计数器，所以我们今天先拿它来开刀）。

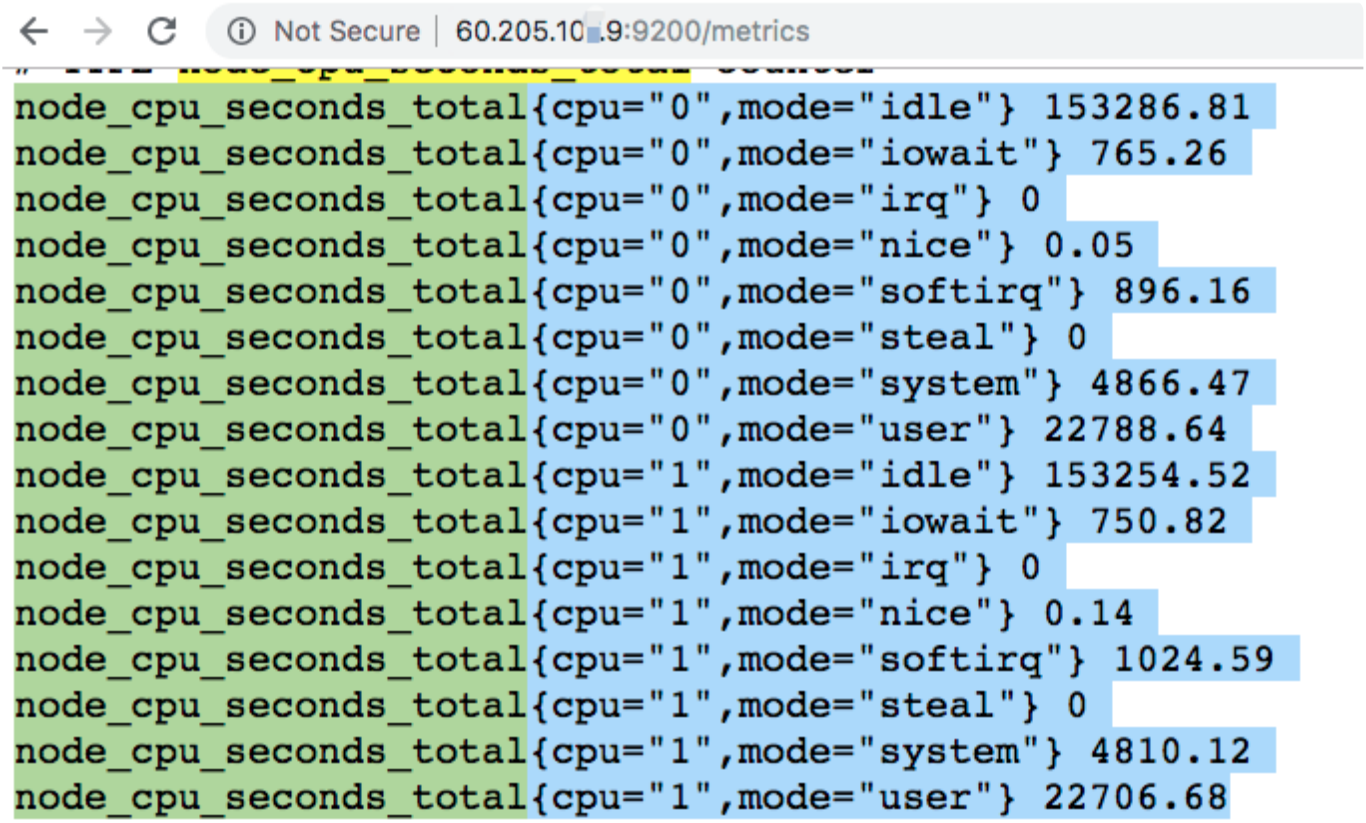
我们先点一下 title 上的 edit，看一下它的 query 语句。

复制代码

```
avg(irate(node_cpu_seconds_total{instance=~"$node",mode="system"}[30m])) by (i
avg(irate(node_cpu_seconds_total{instance=~"$node",mode="user"}[30m])) by (ins
avg(irate(node_cpu_seconds_total{instance=~"$node",mode="iowait"}[30m])) by (i
1 - avg(irate(node_cpu_seconds_total{instance=~"$node",mode="idle"}[30m])) by
```

这些都是从 Prometheus 中取出来的数据，查询语句读了 Prometheus 中`node_cpu_seconds_total`的不同的模块数据。

下面我们来看一下，node_exporter暴露出来的计数器。



这些值和 top 一样，都来自于/proc/目录。下面这张图是 top 数据，我们可以比对一下。

```
top - 15:52:01 up 2 days,  3:18,  1 user,  load average: 2.01, 0.85, 0.56
Tasks:  94 total,   3 running,  91 sleeping,   0 stopped,   0 zombie
%Cpu0  : 36.0 us,  9.6 sy,  0.0 ni, 49.1 id,  3.5 wa,  0.0 hi,  1.8 si,  0.0 st
%Cpu1  : 33.6 us,  7.1 sy,  0.0 ni, 56.6 id,  0.9 wa,  0.0 hi,  1.8 si,  0.0 st
KiB Mem :  3881692 total,   448904 free,  2961312 used,   471476 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   677012 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
12927 root      20   0 2706048 281808   4784 S  21.6  7.3   2:40.73 java
12926 root      20   0 2732332 320800   4968 S  20.7  8.3   8:25.08 java
20610 root      20   0 2729484 343828   4660 S   6.9  8.9   7:19.67 java
12925 root      20   0 2736756 322992   4864 S   6.0  8.3   5:01.96 java
 1837 root      10 -10  131240   6880   2644 S   5.2  0.2  65:17.85 AliYunDun
12924 root      20   0 2722040 324724   4316 S   5.2  8.4   5:03.27 java
10293 mysql     20   0 1566492 386240      0 S   2.6 10.0  23:28.66 mysqld
 6293 root      20   0 2843024  81876   1288 S   1.7  2.1   8:55.18 beam.smp
    9 root      20   0       0      0      0 S   0.9  0.0   2:12.57 rcu_sched
  982 root      20   0   32652   1908    348 S   0.9  0.0   1:21.50 AliYunDunUpdate
 2570 root      20   0  115212  16252   3140 S   0.9  0.4   2:23.42 node_exporter
 4586 rabbitmq  20   0  113176   1580   1316 S   0.9  0.0   0:00.01 rabbitmqctl
    1 root      20   0   51660   2740   1472 S   0.0  0.1   1:29.38 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.03 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:25.48 ksoftirqd/0
```

到此，我们就了解到了操作系统中监控数据的取值逻辑了，也就是从操作系统本身的计数器中取出值来，然后传给 Prometheus，再由 Grafana 中的 query 语句查出相应的数据，最后由 Grafana 展示在界面上。

总结

为什么要解释数据的逻辑呢？

因为最近在工作中遇到一些情况，有人觉得有了 Prometheus+Grafana+Exportor 这样的组合工具之后，基本上都不再用手工执行什么命令了。但我们要了解的是，对于监控平台来说，它取的所有的数据必然是被监控者可以提供的数据，像 node_exporter 这样小巧的监控收集器，它可以获取的监控数据，并不是整个系统全部的性能数据，只是取到了常见的计数器而已。

这些计数器不管是用命令查看，还是用这样炫酷的工具查看，它的值本身都不会变。所以不管是在监控平台上看到的数据，还是在命令行中看到的数据，我们最重要的是要知道含义以及这些值的变化对性能测试和分析的下一步骤的影响。

后面我们将着重来解释这些细节。

问题

最后我问两个问题吧，，第一个问题是，JMeter 如何把数据推送到 Grafana 中呢？另外，同样是监控操作系统的计数器，监控平台中的数据和监控命令中的数据有什么区别？

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　15 | 性能测试场景：如何进行监控设计?

下一篇　17 | CentOS：操作系统级监控及常用计数器解析（上）

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。